Algorithmen und Datenstrukturen

github/bircni

December 27, 2022

Inhaltsverzeichnis

Seite

1 Mathematische Grundlagen

1.1 Reihen

Arithmetische Reihe

- **Allgemeine arithmetische Reihe**: $a_0+(a_0+d)+(a_0+2d)+ \ldots +(a_0+n \cdot d)$

  $$\sum_{i=0}^{n}(a_0+ i \cdot d)=(n+1)(a_0+ d\frac{n}{2})$$

  Beispiel: Summe der ungeraden Zahlen von 1 bis 99, d.h.1 + 3 + 5 + ... + 99:
  $a_0 = 1$
  $d = 2$
  $n = 49$
  Ergebnis: 50 * (1 + 2*49/2) = 2500

- **Gaußsche Summenformel**: 1 + 2 + 3 + ... + *n*, also Summe der natürlichen Zahlen von 1 bis *n*. Dies ist der Spezialfall mit $a_0 = 0$; $d = 1$.

  $$\sum_{i=1}^{n} i=\frac{n(n+1)}{2}$$

  Beispiel: Summe der Zahlen von 1 bis 50:
  50 * 51 / 2 = 1275

wichtig

1.2 Potenzen und Logarithmen

Der Logarithmus ist die Inverse der Potenzfunktion. $\log_a(x) = y \iff a^y = x$
spezielle Logarithmen:
$ld(x) = log_2(x)$, $lg(x) = log_10(x)$, $ln(x) = log_e(x)$

1.3 Notationskonventionen

$\lceil x \rceil$ zur nächsten ganzen Zahl aufrunden
$\lfloor x \rfloor$ zur nächsten ganzen Zahl abrunden
$[a..b] = x | a \leq x \wedge x \leq b$ mit Intervallgrenzen
$]a..b[= x | a < x \wedge x < b$ ohne Intervallgrenzen
$arr[i..k]$ Teilfolge der Elemente von $arr[i]$ bis $arr[k]$

1.4 Grundbegriffe der Graphentheorie

Graphen bestehen aus einer Menge von Knoten und Kanten, die diese verbinden.
Ein Graph ist gerichtet, wenn die Kanten eine Richtung haben.
Für einen Knoten v eines gerichteten Graphen $G = (V, E)$ ist der Eingangsgrad $indeg(v)$ die Anzahl der Kanten, die in v enden, und der Ausgangsgrad $outdeg(v)$ die Anzahl der Kanten, die von v ausgehen.
Ein Zyklus ist ein Weg der bei einem Knoten startet und endet.
Ein gerichteter Graph ist zusammenhängend, wenn es einen Weg zwischen jedem Knotenpaar gibt.
Ein Baum hat einen Knoten als Wurzel, jeder Knoten hat genau einen Vorgänger und ist zusammenhängend.
Ein Knoten ohne Kinder heißt Blatt. Ein leerer Baum hat die Höhe 0. Ein Binärbaum ist ein Baum, dessen Knoten maximal zwei Kinder haben.
Traversierungen: Preorder (WLR), Inorder (LWR), Postorder (LRW)

2 Rekursive Algorithmen

2.1 Prinzip der Rekursion

Ein rekursiver Algorithmus besteht aus einem Basisfall und einem rekursiven Aufruf.
Der rekursive Aufruf muss immer kleiner werden, damit die Rekursion endet.
Die Rekursion kann durch eine Schleife ersetzt werden.

```
  public static double sum_v2(double[] arr) {
  return sum_v2(arr, 0, arr.length-1);
  }
/** Berechnet Summe der Werte von arr[firstIndex..lastIndex] */
  private static double sum_v2(double[] arr, int firstIndex,int lastIndex) {
  if (firstIndex == lastIndex) {
    // zu summierender Bereich besteht nur aus einem Element
    return arr[firstIndex];
  }
  else {
    int mid = (firstIndex + lastIndex) / 2;
    return sum_v2(arr, firstIndex, mid) + sum_v2(arr, mid+1, lastIndex);
  }}
```

2.2 Korrektheit rekursiver Algorithmen

Ein Beweisverfahren ist die Berechnungsinduktion.

Beweis mittels Berechnungsinduktion. Eigenschaft, die nachgewiesen werden soll:

$P((x,n),y) :\Leftrightarrow x^n = y$

d.h. wir wollen zeigen, dass $h(x,n) = x^n$ gilt.

- **Induktionsbasis** (Argumente führen zu keinem rekursiven Aufruf)
  Fall n = 0:
  Ergebnis: $h(x, n) = 1 = x^0$, d.h. Eigenschaft erfüllt
- **Induktionsschritte** (Argumente führen zu rekursiven Aufrufen):
  Wir können als Induktionshypothese für die rekursiven Aufrufe verwenden, dass $h(z_i,k_i) = z_i^{ki}$ gilt.

  Fall n > 0, n gerade:
  Ergebnis laut Programmcode: h(x,n) = y*y, wobei y = h(x, n/2)

  Wir verwenden die Induktionshypothese: es gilt $h(x, n/2) = x^{n/2}$

  somit $h(x,n) = y*y = x^{n/2} * x^{n/2} = x^n$,
  d.h. Eigenschaft in diesem Fall auch für *n* erfüllt.

  Fall n > 0, n ungerade:
  Ergebnis laut Programmcode: h(x,n) = x * h(x,n-1)

  Wir verwenden die Induktionshypothese: es gilt $h(x, n-1) = x^{n-1}$

  Somit $h(x,n) = x * h(x,n-1) = x * x^{n-1} = x^n$,
  d.h. Eigenschaft in diesem Fall auch für *n* erfüllt
- **Induktionsschluss**: somit folgt für alle $n \geq 0$, dass $h(x,n) = x^n$ gilt.

2.3 Rekursive Berechnung der Potenzmenge

Beispiel:
Menge: $M = \{a, b, c\}$
Potenzmenge: $\rho(M) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$

2.3.1 Rekursiver Lösungsansatz

a) in welchen einfach Fällen kann die Lösung direkt angegeben werden?
der einfachste Fall ist die leere Menge $M = \emptyset$
die leere Menge hat nur sich als Teilmenge $\rho(\emptyset) = \{\emptyset\}$
b) Wie können in nicht einfachen Fällen die Teilmengen bestimmt werden?
Sei Menge $M = \{a_1, \ldots, a_{n-1}, a_n\}$ nicht leer $(n \geq 1)$

1. Wir wählen ein Element der Menge, z.B. a_n
2. Es gibt nun zwei Arten von Teilmengen:

   T^+ Teilmengen, die das Element a_n enthalten

   T^- Teilmengen, die das Element a_n nicht enthalten

   Die Menge aller Teilmengen ist die Vereinigung von T^+ und T^-, d.h. $\rho(M) = T^+ \cup T^-$
   Die Menge T^+ kann nun rekursiv berechnet werden, indem wir a_n aus M entfernen und die Potenzmenge von M berechnen.
   Die Menge T^- ist die Potenzmenge von M ohne a_n.
   Die Potenzmenge von M ist also die Vereinigung von T^+ und T^-.

Beispiel: Wenn $M = \{a, b, c\}$
Wähle z.B. c als Element:
T^-: alle Teilmengen ohne c, also alle Teilmengen von $\{a, b\}$
$T^- = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$
T^+: alle Teilmengen mit c, Nimm zu jeder Teilmenge von T^- und füge c hinzu
$T^+ = \{\{c\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$
Insgesamt: $\rho(\{M\}) = T^+ \cup T^- = \{\emptyset, \{a\}, \{b\}, \{a, b\}, \{c\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$

2.3.2 Algorithmischer Ansatz

Falls M leer $(M = \emptyset)$
leere Menge ist die einzige Teilmenge
Falls M nicht leer Wähle ein Element a_n aus M
Berechne Sammlung T^- aller Teilmengen von M ohne a_n (rekursiv)
Berechne Sammlung T^+ aller Teilmengen, die a_n enthalten:
Nimm dazu jede Menge aus T^- und bilde eine neue Menge, indem a_n hinzugefügt wird
Die Menge aller Teilmengen ist die Vereinigung von T^+ und T^-

---

```
private static <E> Set<Set<E>> allSubsets(E[] arr,int maxIndex) {
Set<Set<E>> resultSet = new HashSet<Set<E>>();
if (maxIndex >= 0) {
  // Menge ist nicht leer, waehle letzes Element im gegebenen Bereich
E selected = arr[maxIndex];
  // Bilde rekursive alle Teilmengen ohne selected
Set<Set<E>> resultSet1 = allSubsets(arr, maxIndex - 1);
  // nimm jede dieser Mengen zum Ergebnis hinzu
resultSet.addAll(resultSet1);
  // bilde alle Teilmengen, die selected enthalten
for (Set<E> set1 : resultSet1) {
  // Erzeuge Kopie der Menge aus resultSet1 und nimm gewaehltes Element dazu
```

```
Set<E> set2 = new HashSet<E>(set1);
set2.add(selected);
  // fuege die ergaenzte Kopie zum Ergebnis hinzu
resultSet.add(set2);
}
} else {
  // Menge ist leer. Leere Menge hat nur leere Menge als einzige Teilmenge
Set<E> emptySet = new HashSet<>();
resultSet.add(emptySet);
}
return resultSet;
}
```

2.4 Algorithmenprinzip "Backtracking"

2.4.1 Grundidee "Trial and Error"

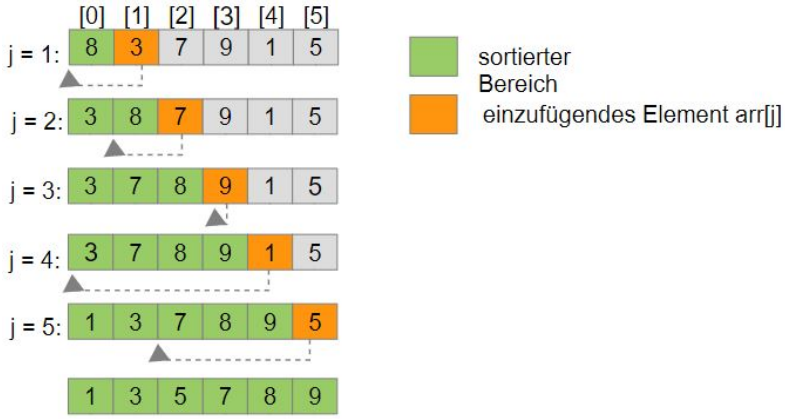
1. Versuche eine Lösung zu finden
2. Wenn die Lösung nicht passt, versuche eine andere Lösung
3. Wenn keine Lösung passt, gehe zurück und versuche eine andere Lösung

3 Analyse von Algorithmen

3.1 Korrektheit

3.1.1 Insertionsort

- Am Anfang des Arrays wird ein sortierter Bereich aufgebaut, in den nach und nach die folgenden Elemente eingefügt werden (deshalb "Insertionsort").
- Am Anfang besteh der sortierte Bereich nur aus dem ersten Element $arr[0]$.
- In jedem Schritt wird ein Element $arr[i]$ aus dem unsortierten Bereich in den sortierten Bereich eingefügt.
- Wenn alle Elemente eingefügt wurden, ist das Array sortiert.



Definition:
Enthält ein Array arr am Anfang die n Elemente $arr[0], arr[1], \ldots, arr[n-1]$, so ist das Array sortiert, wenn gilt:

- **Permutationen:** Die Elemente von arr sind eine Permutattion (Umordnung) der ursprünglichen Elemente von $[0, n-1]$.
- **Monotonie:** Die Elemente von arr sind monoton steigend/fallend sortiert.

Mit **sortiert** ist sofern nicht anders angegeben immer **aufsteigend sortiert** gemeint.

3.2 Komplexität von Algorithmen

Komplexität bezeichnet den Ressourcenverbrauch von Algorithmen. Ressourcen sind dabei die Ausführungszeit und der Speicherbedarf.
Der Ressourcenbedarf hängt von mehreren Faktoren ab:

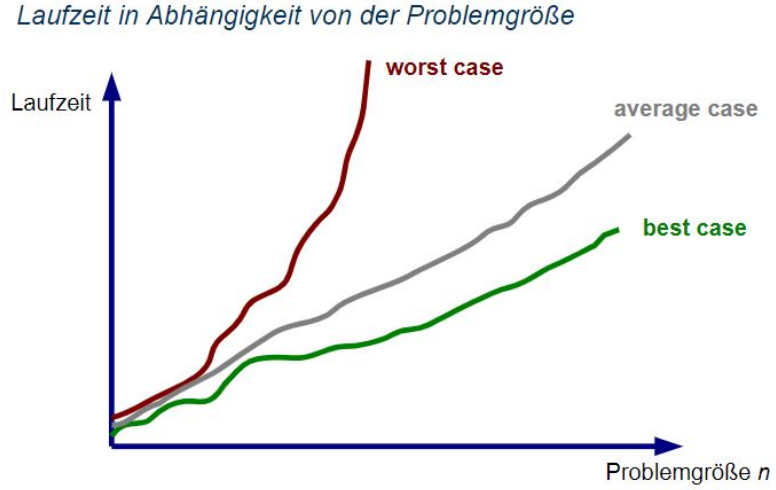
- Umfang der Daten (Größe des Problems)
- Zusammensetzung der Daten (aktuelle Sortierung der Daten)
- Ausführungsgeschwindigkeit und Speicherbedarf bei der Ausführung

3.2.1 Vorgehen bei der Laufzeitanalyse

- Bestimme für jede Anweisung A_j des Programms die Häufigkeit k_j der Ausführung.
- Die Gesamtkosten bei Problemgröße n können dann so zusammengezählt werden: $T(n) = \sum_{A_j}^{n} k_j \cdot c_j$
  wobei k_j die Häufigkeit für die Ausführung von Anweisung A_j ist
  und c_j die Einzel-Ausführungszeit für A_j ist.

3.2.2 Abhängigkeit von der Datenzusammensetzung

Best Case: basiert auf Daten, die im Algorithmus eine minimale Anzahl von Schritten erfordern.
Worst Case: basiert auf Daten, die im Algorithmus eine maximale Anzahl von Schritten erfordern.
Average Case: basiert auf Daten, die im Algorithmus eine durchschnittliche Anzahl von Schritten erfordern.



3.2.3 Häufig verwendete Größenordnungen

- $O(1)$: Konstante Laufzeit - elementare Operationen
- $O(\log n)$: Logarithmische Laufzeit - binäre Suche
- $O(n)$: Lineare Laufzeit - lineare Suche
- $O(n \log n)$: Logarithmische Laufzeit - schnelle Sortieralgorithmen
- $O(n^2)$: Quadratische Laufzeit - einfache Sortieralgorithmen
- $O(n^3)$: Kubische Laufzeit
- $O(c^n)$: Exponentielle Laufzeit
- $O(n!)$: Permutationen berechnen

3.3 Komplexitätsanalyse wohlstrukturierter Algorithmen

Unter wohlstrukturierten Algorithmen versteht man solche, die nur mit Hilfe von Sequenz (nacheinander ausführen), Alternative (Fallunterscheidung) und Iteration (Schleife) definiert sind.

3.3.1 Lineare Suche

Bei der linearen Suche wird ein Array von links nach rechts durchsucht, bis das gesuchte Element gefunden wurde.

```
public String sucheNummer(String suchname) {
  for (int i = 0; i < anzahl; i++) {
    if (liste[i].name.equals(suchname))
    return liste[i].nummer;} // gefunden
return null;} // nicht gefunden
```

3.3.2 Binäre Suche

Bei sortierten Daten kann der Bereich, in dem der gesuchte Schlüssel liegen kann, nach und nach immer wieder halbiert werden. Dieses Verfahren wird **binäre Suche** genannt.

```
private String searchBinRek(String sname, int from, int to){
  if (to < from) {
    return null; //leerer Suchbereich, nichts gefunden
  } else {
  // Mitte des Suchbereichs berechnen
  int middle = (from + to) / 2;
  // Element in der Mitte vergleichen
  int res = suchname.compareTo(liste[middle].name);
    if (res < 0) {
    // in unterer Haelfte weitersuchen
    return searchBinRek(sname, from, middle-1);
    } else if (res > 0) {
    // in oberer Haelfte weitersuchen
    return searchBinRek(sname, middle+1, to);
    } else // (res == 0) {
    // Schluessel gefunden
  return liste[middle].number;
}}}
```

4 Sortierverfahren

4.1 Vergleichsbasierte Sortierverfahren

Interne und Externe Sortierverfahren

- **Interne Sortierverfahren:** alle Datensätze passen in den Hauptspeicher.
- **Externe Sortierverfahren:** nur ein Teil der Datensätze passt in den Hauptspeicher.

Eine Ordnungsrelation heißt **total** wenn sie

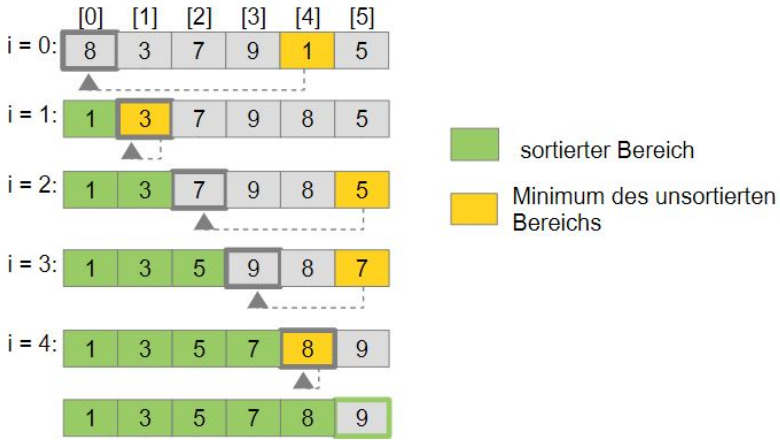
- die Eigenschaften einer Halbordnung erfüllt (reflexiv, transitiv, antisymmetrisch)
  - reflexiv: $a \leq a$
  - transitiv: $a \leq b$ und $b \leq c \Rightarrow a \leq c$
  - antisymmetrisch: $a \leq b$ und $b \leq a \Rightarrow a = b$
- und alle Werte miteinander vergleichbar sind.

Vergleiche über Interface Comparable

```
public interface Comparable {
  int compareTo(Object o) throws ClassCastException;
}
```

wenn $x < y$ dann $x.compareTo(y) < 0$
wenn $x = y$ dann $x.compareTo(y) = 0$
wenn $x > y$ dann $x.compareTo(y) > 0$

4.1.1 Laufzeitkomplexität von Selection-Sort



Worst Case = Best Case = Average Case: $O(n^2)$

4.1.2 Laufzeitkomplexität von Bubble-Sort

Best Case: $O(n)$ - Daten sind schon aufsteigend sortiert
Worst Case: $O(n^2)$ - Daten sind absteigend sortiert
Average Case: $O(n^2)$